Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods

Frédéric Lang

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods

Frédéric Lang

N° 5673

Septembre 2005

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5672--FR+ENG

Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods

Frédéric Lang*

Thème COM — Systèmes communicants
Projet VASY

Rapport de recherche n° 5673 — Septembre 2005 — 21 pages

Abstract: It is desirable to integrate formal verification techniques applicable to different languages. We present Exp.Open 2.0, a new tool of the Cadp verification toolbox which combines several features. First, Exp.Open 2.0 allows to describe concurrent systems as a composition of finite state machines, using either synchronization vectors, or parallel composition, hiding, renaming, and cut operators from several process algebras (Ccs, Csp, Lotos, E-Lotos, μCrl). Second, together with other tools of Cadp, Exp.Open 2.0 allows state space generation and on-the-fly exploration. Third, Exp.Open 2.0 implements on-the-fly partial order reductions to avoid the generation of irrelevant interleavings of independent transitions. Fourth, Exp.Open 2.0 allows to export models towards other tools using interchange formats such as automata networks and Petri nets. Finally, we show some practical applications and measure the efficiency of Exp.Open 2.0 on several benchmarks.

Key-words: Concurrent system, compositional verification, enumerative verification, explicit state verification, labelled transition system, model checking, on-the-fly verification, parallel composition, partial order reduction, process algebra, synchronization vector

A short version of this report is available as "*Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods*", in J. van de Pol, J. Romijn, and G. Smith, editors, Proceedings of the 5th International Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands), November 29-December 2, 2005.

* Frederic.Lang@inria.fr

Unit´e de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
T´el´ephone : +33 4 76 61 52 00 — T´el´ecopie +33 4 76 61 52 52

Exp.Open 2.0 : un outil flexible intégrant réductions d'ordres partiels, vérification compositionnelle et vérification à la volée

Résumé : Il est souhaitable d'intégrer les techniques de vérification formelle applicables à différents langages. Nous présentons Exp.Open 2.0, un nouvel outil de la boîte à outils Cadp, qui combine plusieurs fonctionnalités. Premièrement, Exp.Open 2.0 permet de décrire des systèmes concurrents comme des compositions de machines à états finis, en utilisant des vecteurs de synchronisation et des opérateurs de composition parallèle, de masquage, de renommage et de coupure tirés de plusieurs algèbres de processus (Ccs, Csp, Lotos, E-Lotos, μCrl). Deuxièmement, avec d'autres outils de Cadp, Exp.Open 2.0 permet de générer et d'explorer à la volée les espaces d'états des systèmes décrits. Troisièmement, Exp.Open 2.0 met en œuvre des réductions d'ordre partiel à la volée afin d'éviter la génération d'entrelacements inutiles de transitions indépendantes. Quatrièmement, Exp.Open 2.0 permet d'exporter les modèles vers d'autres outils par le biais de formats d'échange, tels que des réseaux d'automates et des réseaux de Petri. Finalement, nous présentons quelques applications pratiques et nous mesurons l'efficacité de Exp.Open 2.0 sur plusieurs cas d'étude.

Mots-clés : algèbre de processus, composition parallèle, model checking, réduction d'ordre partiel, système concurrent, système de transitions étiquetées, vecteur de synchronisation, vérification à la volée, vérification compositionnelle, vérification de modèle, vérification énumérative

1 Introduction

Enumerative (or *explicit state*) *verification* is a method to check the proper behaviour of safety-critical finite-state systems. It consists in generating the state space systematically (if possible, exhaustively), and in verifying properties by *model checking*, *visual checking*, or *equivalence checking*. For systems involving asynchronous concurrency, the state space is often represented as a *Labelled Transition System* (LTS for short) [47].

A well-known problem with enumerative verification is the combinatorial state explosion, which often occurs as the number of concurrent processes increases. To fight state explosion, several effective techniques have been proposed:

- *Partial order reductions* (e.g., [26, 58, 50, 32, 53, 30, 48]) try to avoid the generation of irrelevant interleavings of independent transitions.
- *On-the-fly verification* (e.g., [16, 15, 38, 33, 46, 45]) consists in performing LTS generation and verification at the same time. This avoids to generate the entire LTS when the verification only requires a part of it.
- *Compositional verification* (e.g., [14, 44, 56, 28, 57, 61, 63, 10, 27, 42, 55, 25, 18]) consists in generating the LTS of each concurrent process first (possibly restricted using constraints derived from its environment [28, 10, 63, 27, 42, 25, 18]), then simplifying these LTSs using abstraction criteria (for instance, label hiding and reductions modulo bisimulations) that preserve the properties under verification, and finally recomposing the reduced LTSs to generate the LTS corresponding to the whole system.

In practice, many software tools have been developed to implement these ideas. Nevertheless, these tools often suffer from several limitations:

- Most tools are often dedicated to one specific input formalism, e.g., Petri nets, communicating automata, or a particular process algebra. On the opposite, a unified tool accepting several input formalisms would be more flexible by combining the expressiveness of different input languages and by having its verification algorithms accessible by a wider community of users.
- Although there exist tools combining two among the three aforementioned verification techniques, such as SPIN [37] (partial order and on-the-fly verification) and ARA [60] (partial order and compositional verification), to our knowledge, combining the three techniques has never been done.

In this report, we present EXP.OPEN 2.0, a new tool that addresses these issues. EXP.OPEN 2.0 is part of CADP [19] (*Construction and Analysis of Distributed Processes*)[1], a toolbox for protocol engineering that offers functionalities ranging from mere interactive simulation up to the most recent verification techniques. EXP.OPEN 2.0 builds upon the existing software components of CADP, especially for handling LTSs.

Earlier versions of CADP contained a tool named EXP.OPEN 1.0, developed in 1995 by L. Mounier (Université Joseph Fourier, Grenoble, France), that combined on-the-fly verification and compositional verification for LOTOS [39]. To develop EXP.OPEN 2.0, we deeply revisited the principles of EXP.OPEN 1.0 and rewrote the tool entirely from scratch to extend its input language, to provide new functionalities, and to support partial order reductions.

[1] `http://www.inrialpes.fr/vasy/cadp`

This report is organized as follows. Section 2 describes inputs of Exp.Open 2.0. Section 3 presents its functionalities. Section 4 presents practical applications and gives experimental results for several applications. Section 5 finally concludes the report.

2 The Exp.Open 2.0 language

2.1 Labelled Transition Systems and composition expressions

The basic concept used by Exp.Open 2.0 is the standard Lts model [47], which consists of a set of *states*, an *initial state*, and a set of *transitions* between states, each transition being labelled by an event of the system. A particular label written τ represents an invisible (or internal) event. The contents of states are not observable.

In practice, a label is represented by a character string. Exp.Open 2.0 does not impose a particular syntax and thus accepts labels from different source languages, such as Ccs [47], Csp [55], Lotos [39], E-Lotos [40], and μCrl [31].

As regards the semantic structure of labels, most languages assume that a label consists of a *gate* (i.e., a port name, a channel name) and a (possibly empty) list of typed values, here called *offers*. For instance, if G is a gate, both labels "G !1 !2" (Lotos notation) and "$G(1,2)$" (μCrl notation) are accepted by Exp.Open 2.0. Labels obtained from Ccs may also start with a *co-action* symbol, generally written '.

Ltss are stored in computer files, using one of the four formats available in Cadp: Bcg (*Binary Coded Graph*), Aldébaran (textual), sequential Fc2, and Seq for transition sequences [20]. Other file formats can be converted into Bcg using the Bcg_Io tool of Cadp.

The input language of Exp.Open 2.0 allows to define compositions of Ltss, named *composition expressions*. Figure 1 presents an extended Bnf describing the abstract syntax of composition expressions. The concrete syntax can be found in [43]. The symbols in italic are the non-terminal and generic terminal symbols. Subscripts are used for the sake of readability, e.g., $B_0, B_1, \ldots$ are occurrences of the same non-terminal B. The symbols "::=", "|", "[", "]", "(", ")", and "..." are meta-symbols: "::=" introduces the definition of a non-terminal symbol, "|" separates alternative clauses, "[ ]" delimit optional clauses, "( )" are used for bracketing as usual, and the infix "..." meta-symbol denotes repetition, e.g., "L_1, ..., L_n" denotes the repetition of $n \geq 0$ symbols separated by commas and "B_1 || ... || B_n" denotes the repetition of $n \geq 0$ symbols separated by ||. All remaining symbols are the terminal symbols, i.e., the keywords (written in bold font, such as **gate**, **all**) and key symbols (written in teletype font, such as "{", "->"). In particular, "[", "]", and "|" are terminal symbols distinct from the meta-symbols "[", "]", and "|".

The generic terminal symbols $L, L', L_0, L_1, \ldots$ represent arbitrary character strings, $n, n_1, n_2, \ldots$ represent arbitrary natural numbers, $S, S_0, S_1, \ldots$ represent Ltss, and $P, P_0, P_1, \ldots$ represent patterns (which will be defined below). The non-terminal symbols $B, B_0, B_1, \ldots$ represent composition expressions, *op* represents binary infix parallel composition operators, and $V, V_0, V_1, \ldots$ represent synchronization vectors.

The semantics of a composition expression is itself an Lts that we define in the following sections.

$$
\begin{array}{lcll}
B & ::= & S_0 & (1)\\
 & | & [\textbf{gate} \mid \textbf{total} \mid \textbf{single} \mid \textbf{multiple}]\ \textbf{rename} & \\
 & & (L_1 \rightarrow L'_1, \ldots, L_n \rightarrow L'_n \mid \textbf{using}\ P_0)\ \textbf{in}\ B_0\ \textbf{end rename} & (2)\\
 & | & [\textbf{gate} \mid \textbf{total} \mid \textbf{partial}]\ \textbf{hide} & \\
 & & ([\textbf{all but}]\ L_1, \ldots, L_n \mid \textbf{using}\ P_0)\ \textbf{in}\ B_0\ \textbf{end hide} & (3)\\
 & | & [\textbf{gate} \mid \textbf{total} \mid \textbf{partial}]\ \textbf{cut} & \\
 & & ([\textbf{all but}]\ L_1, \ldots, L_n \mid \textbf{using}\ P_0)\ \textbf{in}\ B_0\ \textbf{end cut} & (4)\\
 & | & [\textbf{gate} \mid \textbf{label}]\ \textbf{par}\ (\textbf{all} \mid L_1[\#n_1], \ldots, L_m[\#n_m])\ \textbf{in} & \\
 & & [L_1^1, \ldots, L_1^{p_1} \rightarrow]\ B_1\ \texttt{||}\ \ldots\ \texttt{||}\ [L_n^1, \ldots, L_n^{p_n} \rightarrow]\ B_n\ \textbf{end par} & (5)\\
 & | & [\textbf{gate} \mid \textbf{label}]\ \textbf{par}\ V_1, \ldots, V_m\ \textbf{in}\ B_1\ \texttt{||}\ \ldots\ \texttt{||}\ B_n\ \textbf{end par} & (6)\\
 & | & B_1\ op\ B_2 & (7)\\
 & | & B_0\ \texttt{\textbackslash}\ \texttt{\{}\ L_1, \ldots, L_n\ \texttt{\}} & (8)\\
 & | & B_0\ \texttt{[}L_1\ \texttt{/}\ L'_1, \ldots, L_n\ \texttt{/}\ L'_n\ \texttt{]} & (9)\\
 & | & B_0\ \texttt{[[}L_1 \leftarrow L'_1, \ldots, L_n \leftarrow L'_n\ \texttt{]]} & (10)\\
op & ::= & \texttt{|} \mid \texttt{||} \mid \texttt{|||} \mid \texttt{|[}L_1, \ldots, L_n\texttt{]|} & \\
 & | & \texttt{[|}\ L_1, \ldots, L_n\ \texttt{|]} \mid \texttt{[}L_1, \ldots, L_n\ \texttt{||}\ L'_1, \ldots, L'_n\texttt{]} & \\
V & ::= & (L_1 \mid \_) * \ldots * (L_n \mid \_) \rightarrow L_0 &
\end{array}
$$

Figure 1: Abstract syntax of the Exp.Open 2.0 input language.

2.2 Renaming, hiding, and cut operators

Renaming (replacing occurrences of a visible label), hiding (renaming a visible label into τ), and cut[2] (eliminating all transitions with a particular visible label, possibly at the expense of creating unreachable states), are classical notions in process algebras.

For convenience, Exp.Open 2.0 supports the usual notations for these operators found in Ccs and Csp (Rules 8, 9, and 10). Rule 8 represents either Ccs restriction or Csp hiding, which have same syntax but different semantics. Rules 9 and 10 represent Ccs and Csp renaming, respectively. In these three rules, $L_1, L'_1, \ldots, L_n, L'_n$ are simple gates.

Exp.Open 2.0 also supports more expressive operators for renaming, hiding, and cut (Rules 2, 3, and 4), which generalize classical operators in several ways:

- The labels to rename, hide, or cut can be specified either as a list (L_1, ..., L_n), or as a *pattern* ("**using** P_0"), which consists of a reusable list of labels or renaming rules, stored in a separate file for convenience. The latter allows to factor rules used several times, or to isolate complex rules.
- For hiding and cut, the "**all but** L_1, ..., L_n" construct allows to define a set containing all labels but L_1, ..., L_n.
- L_1, ..., L_n can be strings, or regular expressions (following the syntax of the Posix "`regexp`" library) that labels may match using three different semantics: **gate** means that a label matches only if its gate matches a regular expression; **total** means that

[2]Cut is also called *restriction* in Ccs and *encapsulation* in μCrl.

a label matches if it matches a regular expression entirely; and **partial** means that a label matches if it contains a substring that matches a regular expression. As regards renaming, partial matching is refined into two sub-cases: **single** means that only the first occurrence of a substring matching a regular expression is replaced, whereas **multiple** means that all such occurrences are replaced. The **gate** matching is the default, as it corresponds to the semantics found in classical process algebras.

Example 1 *The expression "**hide** G **in** B_0 **end hide**" hides in B_0 every label whose gate is G, such as "G !1 !2" or "$G(1,2)$". The expression "**single rename** "\(.*\) !\(.*\) !\(.*\)" → "\1 !\3 !\2" **in** B_0 **end rename**" permutes two offers in labels, e.g., "G !A !B !C" is renamed into "G !B !A !C"*[3].

2.3 Parallel composition operators

Exp.Open 2.0 contains various parallel composition operators, which can be mixed in the same expression:

- Rule 7 represents the usual binary parallel composition operators of Ccs ("|"), Csp ("[| L_1 , ... , L_n |]" and "[L_1 , ... , L_n || L'_1 , ... L'_n]"), μCrl[4] ("||"), and Lotos ("||", "|||", and "|[L_1 , ... , L_n]|").
- Rule 5 represents the n-ary "graphical" parallel composition operator of E-Lotos [40, 23]. To our knowledge, the Exp.Open 2.0 tool provides the first implementation of this operator in a software tool.
- Rule 6 represents parallel composition using synchronization vectors, inspired from Mec [2] and Fc2 networks [8].

We do not recall in details the semantics of these operators, which are given elsewhere. However, we present an overview of the semantics of the operators of Rules 5 and 6, which are the most general and least known of the Exp.Open 2.0 parallel composition operators.

For these **par** operators, unlike renaming, hiding, and cut, the L_i's and L_i^j's cannot be regular expressions. Nevertheless, Exp.Open 2.0 also extends these operators with a *matching mode*, as follows: **gate** means that the L_i's denote gates and that a label A matches L_i if and only if L_i is the gate of A; **label** means that the L_i's denote full labels and that a label A *matches* L_i if and only if A equals L_i (both gate and offers). The **gate** matching is the default, as it corresponds to the semantics found in classical process algebras.

A *global* state (i.e., a state of the resulting Lts) is a tuple $(s_1, \ldots, s_n)$, where s_i ($i \in 1..n$) is a local state of the corresponding B_i. A global transition is obtained either by synchronization of several local transitions $\{s_i \xrightarrow{A_i} t_i \mid i \in I \subseteq 1..n\}$, or by asynchronous execution of a single local transition $s_i \xrightarrow{A_i} t_i$, where $i \in 1..n$. The destination state of the global transition is obtained by replacing every s_i involved in a local transition by the corresponding t_i, whereas the other local states are not modified.

[3] The symbols "\(" and "\)" are used to delimit sub-expressions. In the right-hand side, the symbol "\n", where n is a number (\1, \2, ...), is substituted by the string matched by the nth delimited sub-expression of the left-hand side.

[4] μCrl parallel composition depends on user-given synchronization rules, whose scope is the whole composition expression. For simplicity, we do not reproduce here the syntax of these rules.

As regards Rule 5, we briefly recall the main features of the "graphical" parallel composition operator (see [23] for a formal description and examples):

- The simplest form, "**par** $L_1, \ldots, L_m$ **in** B_1 || ... || B_n **end par**", is a generalization to n operators of the classical binary parallel composition operators of CSP and LOTOS with forced synchronization on $L_1, \ldots, L_m$. Either one single component evolves asynchronously by executing a transition whose label A does not match any L_i (in such a case, the other components remain in their current state), or all components evolve synchronously by executing transitions whose label A (the same for all components) matches some L_i. In both cases, the resulting global transition is also labelled A. The **all** keyword denotes the set of all gates or labels (depending on the matching mode) but the invisible label τ.
- If some L_i is followed by "#n_i" ($2 \leq n_i \leq n$), then only n_i (instead of n) of the components have to synchronize on L_i. This implements a relaxed form of synchronization named "m among n" synchronization, which is useful to express communication between a subgroup of components, as will be illustrated later.
- If some B_i is preceded by a list, as in "L_i^1 **,** ... **,** $L_i^{p_i} \rightarrow B_i$", then B_i must synchronize on labels matching one of the L_i^j's with all other components also preceded by a list containing L_i^j. This is another form of relaxed synchronization.

Example 2 *In "**par in** $G_{13}, G_{12} \rightarrow B_1$ // $G_{12}, G_{23} \rightarrow B_2$ // $G_{23}, G_{13} \rightarrow B_3$ **end par**", components B_i and B_j ($1 \leq i < j \leq 3$) communicate on gate G_{ij}. In "**par** G_0**#2 in** B_1 // B_2 // B_3 **end par**", components communicate pairwise on gate G_0.*

Rule 6 implements parallel composition using synchronization vectors of the form "(L_1 | _) * ... * (L_n | _) → L_0", whose elements at positions 1..n may be either an L_i (i.e., a gate or a label, depending on the matching mode) or the symbol "_". We define the application of a synchronization vector to the current global state as follows: All components B_i such that the ith element in the vector is an L_i must execute synchronously transitions, such that the label of each transition matches the corresponding L_i (the labels of all transitions must also have the same offers in **gate** matching); the label of the resulting global transition is L_0 (followed by the offers of the synchronizing transitions in **gate** matching). τ-transitions execute asynchronously.

Example 3 *In the expression "**gate par** Snd * Rcv → Com **in** B_1 // B_2 **end par**", transitions of B_1 whose gate is Snd synchronize with transitions of B_2 whose gate is Rcv, provided those transitions have the same offers. The label of the resulting transition consists of the Com gate followed by these offers. In **label** matching instead of **gate**, transitions of B_1 whose label is Snd (gate without offers) would synchronize with transitions of B_2 whose label is Rcv. The label of the resulting transition would be Com, without offers.*

In principle, EXP.OPEN 2.0 allows to freely combine operators originating from different languages, except in case of overloaded symbols that may have different semantics, such as "\" (CCS restriction or CSP hiding) and "||" (LOTOS or μCRL parallel composition). In such cases, a command-line option ("**-ccs**", "**-csp**", etc.) or a specific keyword is needed to indicate to EXP.OPEN 2.0 which language is considered. Command-line options also allow to change syntactic conventions, such as the concrete notation of the invisible label τ

(e.g., `tau`, `i`, `t`) or case-sensitivity (whether or not labels in lower and upper cases are to be considered equal).

The static semantics of Exp.Open 2.0 ensure that synchronization vectors have appropriate length. They also forbid synchronizing, renaming, and cutting τ-transitions, which ensures that bisimulation equivalences (strong, observational, branching, tau*.a, etc.) are congruences for all Exp.Open 2.0 operators [52]. Thus, arbitrary composition expressions of Exp.Open 2.0 can be verified compositionally, for instance by reducing component LTSs separately.

3 State space exploration using Exp.Open 2.0

3.1 Translation into a flat network model

To allow an homogeneous treatment of composition expressions, Exp.Open 2.0 first translates them into a general model, which we call *flat network of* LTS*s* (or simply, flat network).

Flat networks are similar to the **par** operator with synchronization vectors presented in Rule 6 of Figure 1. A flat network is a couple $((S_1, \ldots, S_n), Sync)$ consisting of a vector $(S_1, \ldots, S_n)$ of LTSs, and a set *Sync* of synchronization vectors whose left-hand side (the part to the left of the arrow) have size n. The differences between flat networks and the **par** operator are that synchronization vectors contain "full" labels (instead of gates), including τ, and that flat networks have no nested subterms except LTSs.

Our flat network model is more general than the model used in Exp.Open 1.0, in which synchronization was represented only by vectors of gates (instead of labels), and a global predicate indicating whether a given gate was visible or hidden. This former model allowed to model composition expressions in which a gate was either visible everywhere or hidden everywhere, but not partially visible and partially hidden at the same time, such as in "B || (**hide** G **in** B)" (B containing an occurrence of G), which is legal LOTOS code. This problem imposed that **hide** operators occur at the top-level of expressions only. On the opposite, the whole Exp.Open 2.0 input language can be translated into flat networks without limitations.

A composition expression B is translated into a flat network $(s(B), v(B))$, where $s(B)$ is the vector of all LTSs used in B, in the order of their occurrence (thus LTSs occurring several times in the composition expression also occur several time in the vector), and $v(B)$ is defined recursively as follows:

- For an LTS S (Rule 1 of Figure 1), $v(S) = \{A \rightarrow A \mid A \in \text{labels}(S)\}$.
- For **rename**, **hide**, and **cut** (Rules 2, 3, 4, 8, 9, 10), $v(B_0)$ is computed first. Then, $v(B)$ is obtained by transforming each synchronization vector whose right-hand side matches a renaming, hiding, or cut rule, as follows: For renaming (respectively hiding), the right-hand side of the rule is renamed (respectively hidden) accordingly. For cut, the rule is removed.
- For parallel composition of n sub-expressions $B_1, \ldots, B_n$ (Rules 5, 6, 7), the sets $v(B_1), \ldots, v(B_n)$ are generated first. Their rules are then joined (i.e., their left-hand sides are concatenated and/or extended with an appropriate number of "_" symbols) whenever their respective right-hand sides are synchronizing labels.

Note that the complexity for computing $v(B)$ depends on the number of labels in each of the LTSs in $s(B)$, but not on their number of states and transitions. Therefore, the translation from composition expressions into flat networks is not subject to state explosion.

Example 4 *For the* LOTOS *composition expression "B = (S1 /// S2) /[G]/ S3", where S1, S2, and S3 are* LTSs *and G a list of gates,* $s(B) = (S_1, S_2, S_3)$ *and*

$$
\begin{array}{lllllll}
v(B) = & \{ & A & * \_ & * \_ & \rightarrow A & \mid A \in \mathrm{labels}(S_1), \mathrm{gate}(A) \notin G\} \quad \cup \\
& \{ & \_ & * A & * \_ & \rightarrow A & \mid A \in \mathrm{labels}(S_2), \mathrm{gate}(A) \notin G\} \quad \cup \\
& \{ & \_ & * \_ & * A & \rightarrow A & \mid A \in \mathrm{labels}(S_3), \mathrm{gate}(A) \notin G\} \quad \cup \\
& \{ & A & * \_ & * A & \rightarrow A & \mid A \in \mathrm{labels}(S_1) \cap \mathrm{labels}(S_3), \mathrm{gate}(A) \in G\} \quad \cup \\
& \{ & \_ & * A & * A & \rightarrow A & \mid A \in \mathrm{labels}(S_2) \cap \mathrm{labels}(S_3), \mathrm{gate}(A) \in G\}
\end{array}
$$

EXP.OPEN 2.0 allows to export flat networks into models suitable for various verification tools:

- Petri nets in the "low-level" PEP format, which can be verified using the PEP tool [6] and exported to other Petri net formats.
- Networks of communicating automata in the FC2 format, which can be verified using FC2TOOLS [8] and Jack [1].

3.2 Integration within the OPEN/CAESAR environment

CADP devotes a great importance to modular programming, using well-thought intermediate formats and programming interfaces. EXP.OPEN 2.0 is connected to OPEN/CÆSAR [17], a modular environment for developing on-the-fly exploration algorithms on LTSs.

The OPEN/CÆSAR architecture (see Figure 2) is based on a central language-independent API (*Application Programming Interface*), which allows to explore the states and transitions of an LTS on-the-fly. It describes types that represent labels and states, a function that computes the initial state of the system, and an ITERATE_STATE() function that enumerates the successor transitions of a given state.

This architecture allows an orthogonal separation between the language-dependent compilers (*front-ends*) that translate a particular formalism into a C program implementing the OPEN/CÆSAR API, and the language-independent verification tools (*back-ends*) that operate on the representation of an LTS using the API. Each fromt-end can be combined with any back-end.

CADP includes four front-ends, namely EXP.OPEN 2.0, BCG_OPEN for LTSs in the BCG (*Binary Coded Graphs*) format, CÆSAR [22] for LOTOS [39], and SEQ.OPEN for traces [20]. It also includes several back-ends that provide various functionalities, such as LTS generation, possibly distributed to use the CPU and memory of a set of computers [21], on-the-fly model-checking of regular alternation-free μ-calculus [46], interactive simulation with X-window interface, generation of conformance test suites based on verification technology [41], on-the-fly behavioural comparison of systems modulo various equivalence and preorder relations [5], random execution, deadlock detection, reachability analysis, sequence searching, abstraction of an LTS w.r.t. an interface [42], etc.

EXP.OPEN 2.0 first translates the composition expression given as input into a flat network, and then generates a C program implementing the OPEN/CÆSAR API, which computes the

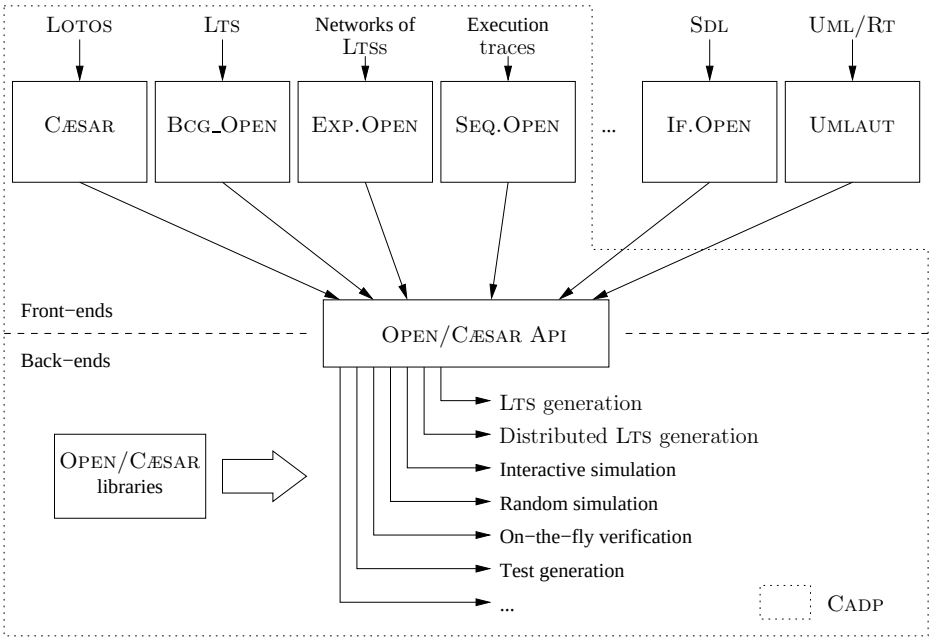

Figure 2: Architecture of OPEN/CÆSAR.

reachable states and transitions of the composition expression. The translation performs careful analysis to reduce the number of bits allocated to represent states, and to optimize speed for the transition function.

3.3 Partial order reductions

Partial order reductions aim at avoiding transition interleavings that are irrelevant for a given class of properties. EXP.OPEN 2.0 implements three partial order reductions, preserving respectively the existence or absence of deadlocks, branching bisimulation [62], and stochastic branching bisimulation [36].

Partial order reduction preserving stochastic branching bisimulation operates on LTSs containing special transitions, called *stochastic*, of the form "**rate** λ", where λ is a positive real. The stochastic transitions express an internal delay in the source state, while the other transitions are immediate if their environment allows their execution. EXP.OPEN 2.0 implements the technique proposed by H. Hermanns [35], which consists in eliminating the stochastic transitions in choice with τ-transitions, the latter being always executable without delay.

To present deadlock and branching preserving partial order reductions, we define the following standard notions derived from the theory of *persistent sets* [26] (of which *stubborn sets* [58] and *ample sets* [50] are variations, see [51] for a survey on persistent set based partial order reductions), and applied to our context:

- A synchronization vector V is *enabled* in a state s if s has a successor obtained by application of V. It is *deterministic* in s if s has exactly one such successor.

- Two synchronization vectors V_1 and V_2 enabled in a state s are *commutative* if the set of states reachable by applying first V_1 then V_2 is the same as that obtained by applying first V_2 then V_1.
- Two synchronization vectors V_1 and V_2 are *independent* in a given state s if 1) V_1 and V_2 are commutative if they are both enabled in s and 2) V_1 (respectively V_2) is enabled in a successor state of s obtained by applying V_2 (respectively V_1) if and only if V_1 (respectively V_2) is enabled in s.
- A set *Sync* of synchronization vectors enabled in a state s is *persistent* in s if, in every state reachable from s by applying only synchronization vectors that do not belong to *Sync*, every synchronization vector that is enabled and does not belong to *Sync* is independent of the synchronization vectors that belong to *Sync*.

Persistent set computation is done by a careful analysis of the synchronization vectors, which we do not detail in this paper. Partial order reduction preserving the presence or absence of deadlocks is done in each reachable state of the system, by applying only the synchronization vectors that belong to the persistent set computed in the current state.

For branching bisimulation, the results of [59, 49, 24] state that, applied to our context, the persistent sets preserving branching bisimulation are those consisting of a single, deterministic synchronization vector, whose right-hand side is the label τ. In the algorithm below, we will only consider persistent sets that have this particular form. Unfortunately, finding such a persistent set is not enough to preserve branching bisimulation — and even weaker relations such as trace equivalence — because one may enter a circuit that prevents enabled synchronization vectors from ever being executed. This problem is known as the *ignoring problem* [51].

Most tools implementing partial order reductions (e.g., SPIN [37], ARA [60], etc.) solve the ignoring problem by detecting circuits in the back-end. A distinctive feature of EXP.OPEN 2.0 is to solve the ignoring problem in the front-end, thus avoiding any modification of verification back-ends, which can thus benefit from partial order reduction for free, independently of the strategy that they use to explore the LTS.

More precisely, the ignoring problem is dealt with in the ITERATE_STATE() function of the OPEN/CÆSAR API. When the ITERATE_STATE() function is called to enumerate the successors of a state s, a persistent set (which contains a single synchronization vector) is searched for. If it exists, this synchronization vector is executed, leading to a new state s'. The algorithm is then repeated, starting in s' instead of s, until reaching a state s'' that either does not have a persistent set or was already visited. In the former case, a single τ-transition from s to s'' is generated. In the latter case, the explored circuit of τ-transitions starting in s'' is just discarded (indeed, all states in a circuit of τ-transitions are branching equivalent) and the algorithm is continued by searching another persistent set in s''.

Note that the intermediate states reachable from s following persistent sets are only stored in memory temporarily by the front-end and never visible by the back-end. They are removed once ITERATE_STATE() returns, to optimize memory consumption. These states may possibly be revisited during subsequent calls to ITERATE_STATE(), but such revisits are not penalizing in practice, mostly due to the fact that persistent set computation is fast in the case of branching bisimulation. This confirms known results [4] about the fine tuning between storing or revisiting states, which were made on the basis of various storage heuristics leading to the conclusion that better verification performance can often be obtained by storing only a little amount of states.

There exist alternative partial order methods preserving branching bisimulation, which are based on τ-confluence reduction [32, 7, 48]. Persistent set methods operate a less general form of τ-confluence reduction than the algorithms presented in [32, 7, 48], but are cheaper in time and memory. In [48], persistent set methods and τ-confluence reduction are combined to reduce LTSs compositionally modulo branching bisimulation, using EXP.OPEN 2.0.

3.4 Refined interface constraints generation

A potential limitation of compositional verification is that, given a system of concurrent processes, generating the LTS of each process separately may lead to state explosion, even though the LTS of the whole system has a tractable size. Indeed, generating the LTS of a process out of its context (i.e., separately from the neighbour processes with which it synchronizes) may lead to explore states that would be unreachable in the global system.

To address this problem, refined compositional verification approaches have been proposed [28, 10, 63, 11, 12, 27, 42, 9, 25], which allow to generate the LTS of a process by taking into account *interface constraints* (also known as *environment constraints* or *context constraints*). These constraints express the behavioural restrictions imposed on each process by the synchronization with its neighbour processes, thus avoiding globally unreachable states and transitions. As regards the choice of appropriate interface constraints, two approaches are possible.

In the first approach, the articles [28, 42] propose that interface constraints may be provided by the user (personal insight of the context). The risk is that these constraints are wrong and thus eliminate states and transitions that would be reachable in the global system. EXP.OPEN 2.0 (together with PROJECTOR 2.0) supports this approach. It checks automatically during the recomposition of the constrained LTS with its environment whether the eliminated states and transitions are indeed unreachable. Otherwise, it reports an error so that the user relaxes its constraints.

The second approach [10, 42] consists in building constraints automatically from the composition expression, for instance by considering a particular LTS in the environment and computing its interactions with the process to restrict. EXP.OPEN 2.0 also implements this approach. Given a flat network, in which are identified an LTS S whose labels are those of a process P to restrict and a set of LTSs $S_1, \ldots, S_n$ corresponding to processes in the environment of P, EXP.OPEN 2.0 computes refined interface constraints consisting of both an LTS S' and a set of labels L representing the potential interactions between $S_1, \ldots, S_n$ and P. S' and L are then used to restrict the LTS corresponding to P using the PROJECTOR 2.0 tool of CADP.

The precise algorithm used by EXP.OPEN 2.0 to generate interface constraints automatically will be detailed in another paper. However, we can briefly indicate the advantages of the proposed approach:

- By operating on flat networks obtained after translation of composition expressions, it can be applied to any of the languages supported by EXP.OPEN 2.0. By constrast, other methods are specific to one single language (e.g., LOTOS [42] or CSP [10]).
- It makes possible to build interface constraints obtained from several processes in the environment of S, even if these processes are distant in the composition expression, because flattening reduces the distance between algebraic terms. Other methods allow to build interface constraints only obtained from one single process.

- In the particular (but frequent) case of nondeterministic synchronization (which is a characteristic of client-server communications), it produces more accurate interface constraints, leading to better state space reductions. For instance, in the LOTOS expression "(B_1 ||| B_2) |[G]| B_3", a G-transition of B_3 can synchronize either with a G-transition of B_1 or with a G-transition of B_2. The same also applies for more complex situations, such as non-deterministic multiway synchronization involving more than two processes, and "m among n" synchronization. Other techniques either forbid such situations using an input language that does not allow nondeterministic synchronization [10] or under-approximate the interactions between B_1 and B_3, and B_2 and B_3, by ignoring the possible synchronizations on G [42]. Instead, EXP.OPEN 2.0 generates interfaces in which every G-transition is duplicated by a τ-transition with same source and target states, which models nondeterministic synchronization.

4 Practical applications and experimental results

As part of CADP, EXP.OPEN 2.0 is widely disseminated and has already been used for significant applications. We can mention for instance a few ones:

- At Eindhoven University of Technology, J. Romijn and S. Vorstenboch used it to verify the *Net Update Protocol* of the draft standard IEEE P1394.1. By combining the compositional techniques of EXP.OPEN 2.0 and the distributed state space construction tool of CADP [21], they managed to generate models of tractable size (up to 28 million states and 487 million transitions).
- At Saarland University, H. Hermanns and S. Johr used the EXP.OPEN 2.0 tool to analyze the performance of a distributed mutual exclusion algorithm. By combining EXP.OPEN 2.0 with the distributed state space construction tool of CADP, they generated a stochastic model with 224 million states and 1,300 million transitions, which was unfortunately too big to fit on a standard 32-bit file system. Using the partial order reduction that preserves stochastic branching reduction, the state space was reduced to 44 million states and 80 million transitions and could be stored in a file on a single machine.
- At INRIA Sophia-Antipolis, E. Madelaine, T. Barros, and L. Henrio used EXP.OPEN 2.0 to compute large synchronization products corresponding to compositions of hierarchical object components [3]. Their work covers dynamic component updates, such as the dynamic replacement of a sub-component.

At least four additional examples of EXP.OPEN 2.0 are available as part of CADP:

- A **distributed summation algorithm**[5] inspired from [29]: The use of "m among n" synchronization allows a nice modeling of the interprocess communications, based on topological constraints encoded using data structures.
- The **ODP trader**[6] inspired from [23]: The use of "m among n" synchronization allows to model communications between arbitrary service providers and service users, which obtain their respective addresses using a separate process, called trader.

[5] `http://www.inrialpes.fr/vasy/cadp/demos/demo_35`
[6] `http://www.inrialpes.fr/vasy/cadp/demos/demo_37`

- The classical **distributed Erathostenes sieve**[7]: It consists of a pipeline of units, each unit blocking every input number that is a multiple of a given number. Table 3 shows experimental data for LTS generation using EXP.OPEN 2.0 from 1 to 20 units, and confirms the effectiveness of partial order reductions.
- The **HAVi leader election protocol for home audio-video networks**[8] [54]: EXP.OPEN 2.0 is used to generate interface constraints automatically. Compared to [54], the LTS corresponding to the largest process was reduced from 400,000 states and 3 million transitions downto 700 states and 2,000 transitions; the memory needed for the whole verification was reduced from 56 MB downto 8.5 MB; the verification time was divided by 10 (from 100 s downto 10 s).

units	without partial order reduction				with partial order reduction			
	states	trans.	time (s)	mem. (MB)	states	trans.	time (s)	mem. (MB)
1	43	59	3.7	2.4	10	9	4.0	2.4
2	159	291	5.1	2.5	10	9	4.9	2.5
3	542	1 233	6.1	2.6	10	9	6.7	2.6
4	1 151	2 909	7.6	2.7	10	9	7.5	2.7
5	3 368	9 831	10.1	2.9	10	9	8.9	2.8
6	12 451	42 423	16.0	3.4	10	9	10.8	3.1
10	166 743	685 951	249.0	11.5	10	9	20.0	5.3
15	—	—	>2h	>113.0	10	9	46.5	17.3
20	—	—	—	—	10	9	99.5	45.8

Figure 3: Generation of configurations of the Erathostenes sieve with and without partial order reduction.

At last, Figure 4 shows that EXP.OPEN 2.0 runs from 2 to 10 times faster and uses 2 times less memory than EXP.OPEN 1.0 on a benchmark consisting of the case studies available in the CADP verification toolbox[9].

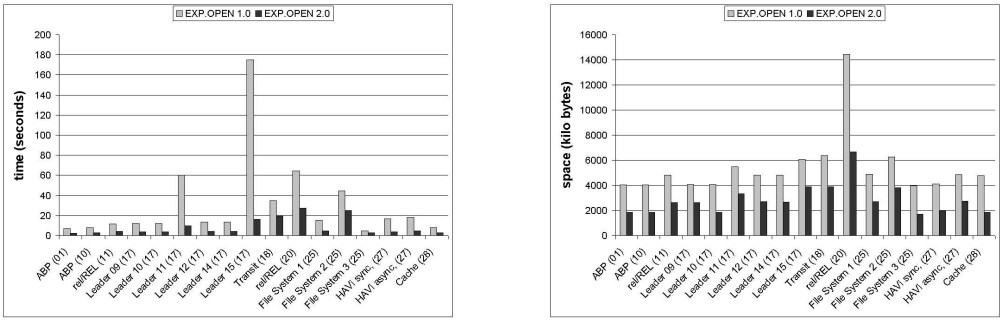

Figure 4: Performance comparisons between EXP.OPEN 1.0 and EXP.OPEN 2.0

[7] http://www.inrialpes.fr/vasy/cadp/demos/demo_36
[8] http://www.inrialpes.fr/vasy/cadp/demos/demo_27
[9] http://www.inrialpes.fr/vasy/cadp/demos

5 Conclusion

In this report, we presented the new Exp.Open 2.0 tool, which has been available in Cadp since August 2004.

While other tools allowing to compute synchronization products are either specific to one language (e.g., Ara-Lotos [60] or Exp.Open 1.0) or implement a single low-level parallel composition operator (e.g., Mec synchronization vectors [2], Fc2 networks [8], Tvt [34], modular Petri nets [13]), Exp.Open 2.0 combines both synchronization vectors [2, 8] and operators taken from several languages, namely Ccs [47], Csp [55], Lotos [39], μCrl [31], and E-Lotos [40]. To our knowledge, Exp.Open 2.0 provides the first implementation of the "graphical" parallel composition operator [23] of E-Lotos, which supports "m among n" synchronization in particular.

Exp.Open 2.0 combines several verification techniques in order to fight combinatorial state explosion effectively. Together with other tools of Cadp, Exp.Open 2.0 allows to generate (possibly using the memory and Cpu of several computers) and explore on-the-fly (for interactive simulation, verification of temporal logics, behavioural equivalence checking, etc.) the Lts of a composition expression. Generation and exploration can be combined with several partial order reductions preserving deadlocks, branching bisimulation, or stochastic branching bisimulation. In addition, Exp.Open 2.0 implements an algorithm to generate interface constraints for compositional verification automatically.

Exp.Open 2.0 has been used for various applications with Lotos and Cadp, which allowed to show its effectiveness. As regards future work, Exp.Open 2.0 could be combined with other languages and tools. Experiments with the μCrl toolset are under way in the framework of the Senva collaboration between Inria and Cwi.

Acknowledgements

The author thanks J. van de Pol for his constructive feedback about the Exp.Open 2.0 tool, for the time he took to proof read the manual page [43], and for helping us to correct minor errors about the handling of μCrl labels. The author is also grateful to H. Garavel for many advices during the development of Exp.Open 2.0 and for his constructive remarks on this report.

References

[1] A bird's eye view of JACK (Just Another Concurrency Kit). Available online at `http://fmt.isti.cnr.it/jack/OLD_JACK_PAGES/JACK/structure.html`.

[2] André Arnold. MEC: A System for Constructing and Analysing Transition Systems. In Joseph Sifakis, editor, *Proceedings of the 1st Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, volume 407 of *Lecture Notes in Computer Science*, pages 117–132. Springer Verlag, June 1989.

[3] T. Barros, L. Henrio, and E. Madelaine. Behavioural Models for Hierarchical Components, 2005. Submitted to the 12th International SPIN Workshop on Model Checking of Software.

[4] G. Behrmann, K.G. Larsen, and R. Pelánek. To Store or Not to Store. In *Proceedings of the 15th International Conference on Computer Aided Verification CAV'2003 (Boulder, Colorado, USA)*, volume 2275 of *Lecture Notes in Computer Science*, 2003.

[5] Damien Bergamini, Nicolas Descoubes, Christophe Joubert, and Radu Mateescu. BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking. In Nicolas Halbwachs and Lenore Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2005 (Edinburgh, Scotland, UK)*, volume 3440 of *Lecture Notes in Computer Science*, pages 581–585. Springer Verlag, April 2005.

[6] E. Best, J. Esparza, B. Grahlmann, S. Melzer, S. Römer, and F. Wallner. The PEP verification system. In *Proceedings of FEmSys'97*, 1997.

[7] Stefan Blom and Jaco van de Pol. State Space Reduction by Proving Confluence. In *Computer Aided Verification 2002*, volume 2404 of *Lecture Notes in Computer Science*, 2002.

[8] Amar Bouali, Annie Ressouche, Valérie Roy, and Robert de Simone. The Fc2Tools set: a Toolset for the Verification of Concurrent Systems. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*. Springer Verlag, August 1996.

[9] K. H. Cheung. *Compositional Analysis of Complex Distributed Systems*. PhD thesis, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.

[10] S. C. Cheung and J. Kramer. Enhancing Compositional Reachability Analysis with Context Constraints. In *Proceedings of the 1st ACM SIGSOFT International Symposium on the Foundations of Software Engineering (Los Angeles, CA, USA)*, pages 115–125. ACM Press, December 1993.

[11] S. C. Cheung and J. Kramer. Compositional Reachability Analysis of Finite-State Distributed Systems with User-Specified Constraints. In *Proceedings of the 3rd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (Washington, DC, USA)*, pages 140–150. ACM Press, October 1995.

[12] S. C. Cheung and J. Kramer. Context Constraints for Compositional Reachability. *ACM Transactions on Software Engineering Methodology TOSEM*, 5(4):334–377, October 1996.

[13] S. Christensen and L. Petrucci. Modular State Space Analysis of Coloured Petri Nets. In G. de. Michelis and M. Diaz, editors, *Proceedings of the 16th International Conference on Application and Theory of Petri Nets*, volume 935 of *Lecture Notes in Computer Science*, 1995.

[14] Jean-Claude Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), May 1988.

[15] Jean-CLaude Fernandez, Claude Jard, Thierry Jéron, and Laurent Mounier. "On the Fly" Verification of Finite Transition Systems. *Formal Methods in System Design*, 1992.

[16] Jean-Claude Fernandez and Laurent Mounier. Verifying Bisimulations "On the Fly". In Juan Quemada, José Manas, and Enrique Vázquez, editors, *Proceedings of the 3rd International Conference on Formal Description Techniques FORTE'90 (Madrid, Spain)*. North-Holland, November 1990.

[17] Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84, Berlin, March 1998. Springer Verlag. Full version available as INRIA Research Report RR-3352.

[18] Hubert Garavel and Frédéric Lang. SVL: a Scripting Language for Compositional Verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea)*, pages 377–392. IFIP, Kluwer Academic Publishers, August 2001. Full version available as INRIA Research Report RR-4223.

[19] Hubert Garavel, Frédéric Lang, and Radu Mateescu. An Overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, August 2002. Also available as INRIA Technical Report RT-0254 (December 2001).

[20] Hubert Garavel and Radu Mateescu. SEQ.OPEN: A Tool for Efficient Trace-Based Verification. In Susanne Graf and Laurent Mounier, editors, *Proceedings of the 11th International SPIN Workshop on Model Checking of Software SPIN'2004 (Barcelona, Spain)*, volume 2989 of *Lecture Notes in Computer Science*, pages 150–155. Springer Verlag, April 2004.

[21] Hubert Garavel, Radu Mateescu, and Irina Smarandache. Parallel State Space Construction for Model-Checking. In Matthew B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software SPIN'2001 (Toronto, Canada)*, volume 2057 of *Lecture Notes in Computer Science*, pages 217–234, Berlin, May 2001. Springer Verlag. Revised version available as INRIA Research Report RR-4341 (December 2001).

[22] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394. IFIP, North-Holland, June 1990.

[23] Hubert Garavel and Mihaela Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In Jianping Wu, Qiang Gao, and Samuel T. Chanson, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'99 (Beijing, China)*, pages 185–202. IFIP, Kluwer Academic Publishers, October 1999.

[24] R. Gerth, R. Kuiper, W. Penczek, and D. Peled. A Partial Order Approach to Branching Time Logic Model Checking. *Information and Computation*, 150(2):132–152, 1999. A short version of this paper was previously published at the Third Israel Symposium on Theory of Computing and Systems ISTCS 1995.

[25] D. Giannakopoulou. *Model Checking for Concurrent Software Architectures*. PhD thesis, Imperial College of Science, Technology and Medicine — University of London — Department of Computer Science, January 1999.

[26] Patrice Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 2nd Workshop on Computer-Aided Verification (Rutgers, New Jersey, USA)*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 321–340. AMS-ACM, June 1990.

[27] S. Graf, B. Steffen, and G. Lüttgen. Compositional Minimisation of Finite State Systems using Interface Specifications. *Formal Aspects of Computation*, 8(5):607–616, September 1996.

[28] Susanne Graf and Bernhard Steffen. Compositional Minimization of Finite State Systems. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 2nd Workshop on Computer-Aided Verification (Rutgers, New Jersey, USA)*, volume 531 of *Lecture Notes in Computer Science*, pages 186–196. Springer Verlag, June 1990.

[29] J.F. Groote, F. Monin, and J. Springintveld. A Computer Checked Algebraic Verification of a Distributed Summation Algorithm. Computer Science Report 97/14, Department of Mathematics and Computer Science, Eindhoven University of Technology, 1997.

[30] J.F. Groote and J. van de Pol. State Space Reduction using Partial τ-Confluence. In Mogens Nielsen and Branislav Rovan, editors, *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science MFCS'2000 (Bratislava, Slovakia)*, volume 1893 of *Lecture Notes in Computer Science*, pages 383–393, Berlin, August 2000. Springer Verlag. Also available as CWI Technical Report SEN-R0008, Amsterdam, March 2000.

[31] J.F. Groote and A. Ponse. Syntax and semantics of μ-CRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes, Workshops in Computing*, pages 26–62, 1995.

[32] J.F. Groote and M.P.A. Sellink. Confluence for process verification. *Theoretical Computer Science*, 170(1–2):47–81, December 1996.

[33] H. Hansen, W. Penczek, and A. Valmari. Stuttering-Insensitive Automata for On-the-fly Detection of Livelock Properties. In *7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems*, volume 66 of *Electronic Notes in Theoretical Computer Science*, 2002.

[34] Henri Hansen, Heikki Virtanen, and Antti Valmari. Merging State-Based and Action-Based Verification. In *Proceedings of the Third International Conference on Application of Concurrency to System Design*. IEEE Computer Society, 2003.

[35] Holger Hermanns. *Interactive Markov Chains and the Quest for Quantified Quality*, volume 2428 of *LNCS*. Springer Verlag, 2002.

[36] Holger Hermanns and Markus Siegle. Bisimulation Algorithms for Stochastic Process Algebras and their BDD-based Implementation. In Joost-Pieter Katoen, editor, *Proceedings of the 5th International AMAST Workshop ARTS'99 (Bamberg, Germany)*, volume 1601 of *Lecture Notes in Computer Science*, pages 244–265. Springer Verlag, May 1999.

[37] G. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[38] G.J. Holzmann. On-The-Fly Model Checking. *ACM Computing Surveys*, 28(4), 1996.

[39] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1989.

[40] ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization — Information Technology, Genève, September 2001.

[41] T. Jéron and P. Morel. Test generation derived from model-checking. In N. Halbwachs and D. Peled, editors, *Proceedings of the Conference on Computer-Aided Verification CAV'99 (Trento, Italy)*, volume 1633 of *Lecture Notes in Computer Science*, pages 108–122. Springer Verlag, July 1999.

[42] Jean-Pierre Krimm and Laurent Mounier. Compositional State Space Generation from LOTOS Programs. In Ed Brinksma, editor, *Proceedings of TACAS'97 Tools and Algorithms for the Construction and Analysis of Systems (University of Twente, Enschede, The Netherlands)*, volume 1217 of *Lecture Notes in Computer Science*, Berlin, April 1997. Springer Verlag. Extended version with proofs available as Research Report VERIMAG RR97-01.

[43] Frédéric Lang. The Exp.Open 2.0 manual page, 2004. Available online at `http://www.inrialpes.fr/vasy/cadp/man/exp.open.html`.

[44] J. Malhotra, S. A. Smolka, A. Giacalone, and R. Shapiro. A Tool for Hierarchical Design and Simulation of Concurrent Systems. In *Proceedings of the BCS-FACS Workshop on Specification and Verification of Concurrent Systems (Stirling, Scotland)*, pages 140–152, Swinton, UK, July 1988. British Computer Society.

[45] Radu Mateescu. A Generic On-the-Fly Solver for Alternation-Free Boolean Equation Systems. In Hubert Garavel and John Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2003 (Warsaw, Poland)*, volume 2619 of *Lecture Notes in Computer Science*, pages 81–96. Springer Verlag, April 2003. Full version available as INRIA Research Report RR-4711.

[46] Radu Mateescu and Mihaela Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, March 2003.

[47] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[48] Gordon Pace, Frédéric Lang, and Radu Mateescu. Calculating τ-Confluence Compositionally. In Jr Warren A. Hunt and Fabio Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification CAV'2003 (Boulder, Colorado, USA)*, volume 2725 of *Lecture Notes in Computer Science*, pages 446–459. Springer Verlag, July 2003. Full version available as INRIA Research Report RR-4918.

[49] D. Peled. Partial Order Reduction: Linear and Branching Temporal Logics and Process Algebras. In Peled et al. [51].

[50] D.A. Peled. Combining partial order reduction with on-the-fly model-checking. In *Computer Aided Verification 1994*, volume 818 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[51] D.A. Peled, V.R. Pratt, and G.J. Holzmann, editors. *Proceedings of the Workshop on Partial Order Methods in Verification*, volume 29 of *Dimacs Series in Discrete Mathematics*, 1997.

[52] Jaco van de Pol. Proof using the PVS theorem prover that bisimulations are congruences for synchronization vectors that do not rename, cut, nor synchronize τ-transitions, 2003. Personal communication.

[53] Y.S. Ramakrishna and S.A. Smolka. Partial-Order Reduction in the Weak Modal Mu-Calculus. In A. Mazurkiewicz and J. Winkowski, editors, *Proceedings of the 8th International Conference on Concurrency Theory CONCUR'97*, volume 1243 of *Lecture Notes in Computer Science*, pages 5–24. Springer Verlag, 1997.

[54] Judi Romijn. Model Checking the HAVi Leader Election Protocol. Technical Report SEN-R9915, CWI, Amsterdam, The Netherlands, June 1999. submitted to Formal Methods in System Design.

[55] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[56] K. K. Sabnani, A. M. Lapone, and M. U. Uyar. An Algorithmic Procedure for Checking Safety Properties of Protocols. *IEEE Transactions on Communications*, 37(9):940–948, September 1989.

[57] K. C. Tai and V. Koppol. Hierarchy-Based Incremental Reachability Analysis of Communication Protocols. In *Proceedings of the IEEE International Conference on Network Protocols (San Francisco, CA)*, pages 318–325, Piscataway, NJ, October 1993. IEEE Press.

[58] A. Valmari. A Stubborn Attack on State Explosion. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 2nd Workshop on Computer-Aided Verification (Rutgers, New Jersey, USA)*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 25–42. AMS-ACM, June 1990.

[59] A. Valmari. Stubborn Set Methods for Process Algebras. In Peled et al. [51].

[60] A. Valmari, J. Kemppainen, M. Clegg, and M. Levanto. Putting Advanced Reachability Analysis Techniques Together: the "ARA" Tool. In *Proceedings of the First International Symposium of Formal Methods Europe FME '93*, volume 670 of *Lecture Notes in Computer Science*, pages 597–616. Springer-Verlag, 1993.

[61] Antti Valmari. Compositional State Space Generation. In *Proceedings of Advances in Petri Nets*, volume 674 of *Lecture Notes in Computer Science*, pages 427–457. Springer Verlag, 1993.

[62] R. J. van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989. Also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.

[63] W. J. Yeh. *Controlling State Explosion in Reachability Analysis*. PhD thesis, Software Engineering Research Center (SERC) Laboratory, Purdue University, December 1993. Technical Report SERC-TR-147-P.

Unit´e de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unit´e de recherche INRIA Futurs : Parc Club Orsay Universit´e - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unit´e de recherche INRIA Lorraine : LORIA, Technopˆole de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unit´e de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unit´e de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unit´e de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399